Lecture 7: Linear Algebra Algorithms

Jack Dongarra, U of Tennessee

Algorithms and Architecture
° The key to performance is to understand the algorithm and architecture interaction.
° A significant improvement in performance can be obtained by matching algorithm to the architecture or vice versa.

Algorithm Issues
° Use of memory hierarchy
° Algorithm pre-fetching
° Loop unrolling
° Simulating higher precision arithmetic

Blocking
° TLB Blocking - minimize TLB misses
° Cache Blocking - minimize cache misses
° Register Blocking - minimize load/stores
° The general idea of blocking is to get the information to a high-speed storage and use it multiple times so as to amortize the cost of moving the data
° Cache blocking reduced traffic between memory and cache
° Register blocking reduces traffic between cache and CPU

Loop Unrolling
° Reduces data dependency delay
° Exploits multiple functional units and quad load/stores effectively.
° Minimizes load/stores
° Reduces loop overheads
° Gives more flexibility to compiler in scheduling
° Facilitates algorithm pre-fetching.
° What about vector computing?

Performance Numbers on RISC Processors
° Using Linpack Benchmark

<table>
<thead>
<tr>
<th>Machine</th>
<th>MHz</th>
<th>Linpack n=100</th>
<th>Linpack n=1000</th>
<th>Axel n=1000</th>
<th>Peak MFlop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC Alpha</td>
<td>588</td>
<td>460 (40%)</td>
<td>847 (74%)</td>
<td>1150</td>
<td></td>
</tr>
<tr>
<td>IBM RS6K</td>
<td>375</td>
<td>409 (27%)</td>
<td>1236 (82%)</td>
<td>1500</td>
<td></td>
</tr>
<tr>
<td>HP PA</td>
<td>440</td>
<td>375 (21%)</td>
<td>1047 (59%)</td>
<td>1760</td>
<td></td>
</tr>
<tr>
<td>SUN Ultra</td>
<td>450</td>
<td>208 (23%)</td>
<td>697 (67%)</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>SGI Origin2K</td>
<td>300</td>
<td>173 (29%)</td>
<td>553 (92%)</td>
<td>600</td>
<td></td>
</tr>
<tr>
<td>Intel PII Xeon</td>
<td>450</td>
<td>98 (22%)</td>
<td>295 (66%)</td>
<td>450</td>
<td></td>
</tr>
<tr>
<td>Cray T90</td>
<td>454</td>
<td>705 (39%)</td>
<td>1683 (98%)</td>
<td>1800</td>
<td></td>
</tr>
<tr>
<td>Cray C90</td>
<td>238</td>
<td>387 (41%)</td>
<td>902 (95%)</td>
<td>952</td>
<td></td>
</tr>
<tr>
<td>Cray Y-MP</td>
<td>166</td>
<td>161 (48%)</td>
<td>324 (97%)</td>
<td>333</td>
<td></td>
</tr>
</tbody>
</table>
What’s Wrong With Speedup $T_1/T_p$?
- Can lead to very false conclusions.
- Speedup in isolation without taking into account the speed of the processor is unrealistic and pointless.
- Speedup over what?
  - $T_1/T_p$
    - There is usually no doubt about $T_1$
    - Often considerable dispute over the meaning of $T_p$
      - Serial code? Same algorithm?

Speedup
- Can be used to:
  - Study, in isolation, the scaling of one algorithm on one computer.
  - As a dimensionless variable in the theory of scaling.
- Should not be used to compare:
  - Different algorithms on the same computer
  - The same algorithm on different computers.
  - Different interconnection structures.

Linpack - an Example?

Strassen’s Algorithm
- The count of arithmetic operations is:

<table>
<thead>
<tr>
<th></th>
<th>Mat</th>
<th>Add</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular</td>
<td>4</td>
<td>4</td>
<td>$2^n + O(n^2)$</td>
</tr>
<tr>
<td>Strassen</td>
<td>7</td>
<td>18</td>
<td>$4.7 n^{\log_2 7} + 0(n^2)$</td>
</tr>
</tbody>
</table>

One matrix multiply is replaced by 14 matrix additions.
Strassen’s Algorithm

In reality the use of Strassen’s Algorithm is limited by
- Additional memory required for storing the P matrices.
- More memory accesses are needed.

Outline

- Motivation for Dense Linear Algebra
  - Ax=b: Computational Electromagnetics
  - Ax=c: Quantum Chemistry
- Review Gaussian Elimination (GE) for solving Ax=b
- Optimizing GE for caches on sequential machines
  - using matrix-matrix multiplication (BLAS)
- LAPACK library overview and performance
- Data layouts on parallel machines
  - Parallel matrix-matrix multiplication
  - Parallel Gaussian Elimination
  - ScaLAPACK library overview
  - Eigenvalue problem

Parallelism in Sparse Matrix-vector multiplication

- \( y = A x \), where \( A \) is sparse and \( n \times n \)
- Questions
  - which processors store \( y[i], x[i], \) and \( A[i,j] \)
- which processors compute
  - \( y[i] = \text{sum (from 1 to n)} A[i,j] * x[j] \)
  - \( (\text{row i of A}) \cdot x \)  — a sparse dot product
- Partitioning
  - Partition index set \( \{1, ..., n\} = N_1 \cup N_2 \cup ... \cup N_p \)
  - For all \( i \in N_k \), Processor \( k \) stores \( y[i], x[i], \) and row \( i \) of \( A \)
  - For all \( i \in N_k \), Processor \( k \) computes \( y[i] = (\text{row i of A}) \cdot x \)
  - “owner computes” rule: Processor \( k \) compute the \( y[i] \)s it owns
- Goals of partitioning
  - balance load (how is load measured?)
  - balance storage (how much does each processor store?)
  - minimize communication (how much is communicated?)

Graph Partitioning and Sparse Matrices

- Relationship between matrix and graph

A “good” partition of the graph has
- equal (weighted) number of nodes in each part (load and storage balance)
- minimum number of edges crossing between (minimize communication)
- Can reorder the rows/columns of the matrix by putting all the nodes in one partition together

More on Matrix Reordering via Graph Partitioning

- “Ideal” matrix structure for parallelism: (nearly) block diagonal
  - \( p \) (number of processors) blocks
  - few non-zeros outside these blocks, since these require communication

What about implicit methods and eigenproblems?

- Direct methods (Gaussian elimination)
  - Called LU Decomposition, because we factor \( A = L \cdot U \)
  - Future lectures will consider both dense and sparse cases
  - More complicated than sparse-matrix vector multiplication
- Iterative solvers
  - Will discuss several of these in future
    - Jacobi, Successive overrelaxation (SOR), Conjugate Gradients (CG), Multigrid, ...
  - Most have sparse-matrix-vector multiplication in kernel
- Eigenproblems
  - Future lectures will discuss dense and sparse cases
  - Also depend on sparse-matrix-vector multiplication, direct methods
  - Graph partitioning
Partial Differential Equations

PDEs

Examples of such systems include

- Heat flow: Temperature(position, time)
- Diffusion: Concentration(position, time)
- Electrostatic or Gravitational Potential:
  Potential(position)
- Fluid flow: Velocity, Pressure, Density(position, time)
- Quantum mechanics: Wave-function(position, time)
- Elasticity: Stress, Strain(position, time)

Example: Deriving the Heat Equation

Consider a simple problem

- A bar of uniform material, insulated except at ends
- Let \( u(x,t) \) be the temperature at position \( x \) at time \( t \)
- Heat travels from \( x-h \) to \( x+h \) at rate proportional to:

\[
\frac{du(x,t)}{dt} = C \cdot \frac{(u(x-h,t)-u(x+h,t))/h}{h}
\]

As \( h \to 0 \), we get the heat equation:

\[
\frac{du(x,t)}{dt} = \frac{d^2u(x,t)}{dx^2}
\]

Explicit Solution of the Heat Equation

- For simplicity, assume \( C=1 \)
- Discretize both time and position
- Use finite differences with \( u[j,i] \) as the heat at
  - time \( t = i \cdot dt \) (\( i = 0, 1, 2, ... \))
  - position \( x = j \cdot h \) (\( j = 0, 1, ..., N \))
- initial conditions on \( u[j,0] \)
- boundary conditions on \( u[0,i] \) and \( u[N,i] \)
- At each timestep \( i = 0, 1, 2, ... \)

\[
u[j,i+1] = z \cdot u[j-1,i] + (1-2z) \cdot u[j,i] + z \cdot u[j+1,i]
\]

where \( z = \frac{dt}{h^2} \)

This corresponds to

- matrix vector multiply (what is matrix?)
- nearest neighbors on grid

Parallelism in Explicit Method for PDEs

- Partitioning the space (x) into p largest chunks
  - good load balance (assuming large number of points relative to p)
  - minimized communication (only p chunks)

- Generalizes to
  - multiple dimensions
  - arbitrary graphs (sparse matrices)

- Problem with explicit approach
  - numerical instability
  - solution blows up eventually if \( z = \frac{dt}{h^2} > 0.5 \)
  - need to make the timesteps very small when \( h \) is small: \( dt < 0.5 h^2 \)

Instability in solving the heat equation explicitly
Implicit Solution

° As with many (stiff) ODEs, need an implicit method
° This turns into solving the following equation

\[(I + (z/2)T) \cdot u[:,i+1] = (I - (z/2)T) \cdot u[:,i]\]

° Here \(I\) is the identity matrix and \(T\) is:

\[
T = \begin{bmatrix}
2 & -1 & 0 & \cdots & 0 \\
-1 & 2 & -1 & \cdots & 0 \\
0 & -1 & 2 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & -1\
\end{bmatrix}
\]

° I.e., essentially solving Poisson’s equation in 1D

2D Implicit Method

° Similar to the 1D case, but the matrix \(T\) is now

\[
T = \begin{bmatrix}
4 & -1 & 0 & \cdots & 0 \\
-1 & 4 & -1 & \cdots & 0 \\
0 & -1 & 4 & \cdots & 0 \\
\vdots & \ddots & \ddots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & 4\
\end{bmatrix}
\]

° Multiplying by this matrix (as in the explicit case) is simply nearest neighbor computation on 2D grid
° To solve this system, there are several techniques

Algorithms for 2D Poisson Equation with N unknowns

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Serial</th>
<th>PRAM</th>
<th>Memory</th>
<th>#Procs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense LU</td>
<td>N(^2)</td>
<td>N</td>
<td>N(^2)</td>
<td>N(^2)</td>
</tr>
<tr>
<td>Band LU</td>
<td>N(^2)</td>
<td>N</td>
<td>N(^2)</td>
<td>N(^2)</td>
</tr>
<tr>
<td>Jacobi</td>
<td>N(^2)</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Explicit Inv.</td>
<td>N(^2)</td>
<td>log N</td>
<td>N(^2)</td>
<td>N(^2)</td>
</tr>
<tr>
<td>Conj.Grad.</td>
<td>N(^{2\times3})</td>
<td>log N</td>
<td>N(^{2\times3})</td>
<td>N(^{2\times3})</td>
</tr>
<tr>
<td>RB SOR</td>
<td>N(^{2\times3})</td>
<td>N(^{1\times2})</td>
<td>N (\times N)</td>
<td>N (\times N)</td>
</tr>
<tr>
<td>Sparse LU</td>
<td>N(^{2\times3})</td>
<td>N(^{1\times2})</td>
<td>N(^{1\times2})</td>
<td>N(^{1\times2})</td>
</tr>
<tr>
<td>FFT</td>
<td>N(^{2\times3})</td>
<td>N(^{1\times2})</td>
<td>N(^{1\times2})</td>
<td>N(^{1\times2})</td>
</tr>
<tr>
<td>Multigrid</td>
<td>N(^{2\times3})</td>
<td>N(^{1\times2})</td>
<td>N(^{1\times2})</td>
<td>N(^{1\times2})</td>
</tr>
<tr>
<td>Lower bound</td>
<td>N(^{2\times3})</td>
<td>N(^{1\times2})</td>
<td>N(^{1\times2})</td>
<td>N(^{1\times2})</td>
</tr>
</tbody>
</table>

PRAM is an idealized parallel model with zero cost communication

(see next slide for explanation)

Short explanations of algorithms on previous slide

° Sorted in two orders (roughly):
  ° from slowest to fastest on sequential machines
  ° from most general (works on any matrix) to most specialized (works on matrices “like” \(T\))
° Dense LU: Gaussian elimination; works on any \(N\)-by-\(N\) matrix
° Band LU: exploit fact that \(T\) is nonzero only on \(\sqrt{N}\) diagonals nearest main diagonal, so faster
° Jacobi: essentially does matrix-vector multiply by \(T\) in inner loop of iterative algorithm
° Explicit Inv: assume we want to solve many systems with \(T\), so we can precompute and store \(inv(T)\) for free, and just multiply by it
° It's still expensive!
° Conjugate Gradient: uses matrix-vector multiplication, like Jacobi, but exploits mathematical properties of \(T\) that Jacobi does not
° Run-Black SOR (Successive Overrelaxation): Variation of Jacobi that exploits yet different mathematical properties of \(T\)
° Multigrid: also works on matrices like \(T\), that come from elliptic PDEs
° Lower Bound: serial (time to print answer); parallel (time to combine \(N\) inputs)

Composite mesh from a mechanical structure

Converting the mesh to a matrix
Effects of Ordering Rows and Columns on Gaussian Elimination

Irregular mesh: NASA Airfoil in 2D (direct solution)

Irregular mesh: Tapered Tube (multigrid)

Example of Proteron meshes

Adaptive Mesh Refinement (AMR)

*Adaptive mesh around an explosion
*John Bell and Phil Colella at LBL (see class web page for URL)
*Goal of Titanium is to make these algorithms easier to implement in parallel

Computational Electromagnetics

- Developed during 1980s, driven by defense applications
- Determine the RCS (radar cross section) of airplane
- Reduce signature of plane (stealth technology)
- Other applications are antenna design, medical equipment
- Two fundamental numerical approaches:
  - MOM methods of moments (frequency domain), and
  - Finite differences (time domain)

Computational Electromagnetics

- Discretize surface into triangular facets using standard modeling tools
- Amplitude of currents on surface are unknowns

- Integral equation is discretized into a set of linear equations

Image: NW Univ. Comp. Electromagnetics Laboratory  http://nueml.ece.nwu.edu/
After discretization the integral equation has the form

\[ A x = b \]

where

- \( A \) is the (dense) impedance matrix,
- \( x \) is the unknown vector of amplitudes, and
- \( b \) is the excitation vector.

(see Cwik, Patterson, and Scott, Electromagnetic Scattering on the Intel Touchstone Delta, IEEE Supercomputing '92, pp 538 - 542)

The main steps in the solution process are

- **Fill**: computing the matrix elements of \( A \)
- **Factor**: factoring the dense matrix \( A \)
- **Solve**: solving for one or more excitations \( b \)
- **Field Calc**: computing the fields scattered from the object

### Analysis of MOM for Parallel Implementation

<table>
<thead>
<tr>
<th>Task</th>
<th>Work</th>
<th>Parallelism</th>
<th>Parallel Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fill</td>
<td>( O(n^2) )</td>
<td>embarrassing</td>
<td>low</td>
</tr>
<tr>
<td>Factor</td>
<td>( O(n^3) )</td>
<td>moderately diff.</td>
<td>very high</td>
</tr>
<tr>
<td>Solve</td>
<td>( O(n^2) )</td>
<td>moderately diff.</td>
<td>high</td>
</tr>
<tr>
<td>Field Calc.</td>
<td>( O(n) )</td>
<td>embarrassing</td>
<td>high</td>
</tr>
</tbody>
</table>

### Results for Parallel Implementation on Delta

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (hours)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fill</td>
<td>9.20</td>
</tr>
<tr>
<td>Factor</td>
<td>8.25</td>
</tr>
<tr>
<td>Solve</td>
<td>2.17</td>
</tr>
<tr>
<td>Field Calc.</td>
<td>0.12</td>
</tr>
</tbody>
</table>

The problem solved was for a matrix of size 48,672. (The world record in 1991.)

### Current Records for Solving Dense Systems

<table>
<thead>
<tr>
<th>Year</th>
<th>System Size</th>
<th>Machine</th>
<th># Procs</th>
<th>Gflops (Peak)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950’s</td>
<td>( O(100) )</td>
<td>Intel Paragon</td>
<td>6768</td>
<td>281 (338)</td>
</tr>
<tr>
<td>1995</td>
<td>129,600</td>
<td>Intel ASCI Red</td>
<td>7264</td>
<td>1056 (1453)</td>
</tr>
<tr>
<td>1996</td>
<td>215,000</td>
<td>Intel ASCI Red</td>
<td>1152</td>
<td>830 (1258)</td>
</tr>
<tr>
<td>1998</td>
<td>148,000</td>
<td>Cray T3E</td>
<td>1127 (1786)</td>
<td></td>
</tr>
<tr>
<td>1998</td>
<td>235,000</td>
<td>Intel ASCI Red</td>
<td>1338 (1830)</td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>374,000</td>
<td>SGI ASCI Blue</td>
<td>5040</td>
<td>1608 (2520)</td>
</tr>
<tr>
<td>1999</td>
<td>362,880</td>
<td>Intel ASCI Red</td>
<td>9632</td>
<td>2379 (3207)</td>
</tr>
</tbody>
</table>


### Computational Chemistry

° Seek energy levels of a molecule, crystal, etc.
- Solve Schrödinger’s Equation for energy levels = eigenvalues
- Discretize to get \( A x = \beta x \), solve for eigenvalues \( \beta \) and eigenvectors \( x \)
- A and B large, symmetric or Hermitian matrices (B positive definite)
- May want some or all eigenvalues/eigenvectors

° MP-Quest (Sandia NL)
- Si and sapphire crystals of up to 3072 atoms
- Local Density Approximation to Schrödinger Equation
- A and B up to \( n=40000 \), Hermitian
- Need all eigenvalues and eigenvectors
- Need to iterate up to 20 times (for self-consistency)

° Implemented on Intel ASCI Red
- 9200 Pentium Pro 200 processors (400 Duals, a CLUMP)
- Overall application ran at 605 Gflops (out of 1800 Gflops peak),
- Eigensolver ran at 684 Gflops
- [www.cs.berkeley.edu/~stanley/gbell/index.html](http://www.cs.berkeley.edu/~stanley/gbell/index.html)
- Runner-up for Gordon Bell Prize at Supercomputing 98
Review of Gaussian Elimination (GE) for solving $Ax=b$

- Add multiples of each row to later rows to make $A$ upper triangular
- Solve resulting triangular system $Ux = c$ by substitution

Refine GE Algorithm (1)

- **Initial Version**
  - for each column $i$
    - zero it out below the diagonal by adding multiples of row $i$ to later rows
    - for each row $j$ below row $i$
      - add a multiple of row $i$ to row $j$
      - for $k = i$ to $n$
        - $A(j,k) = A(j,k) - (A(j,i)/A(i,i)) \cdot A(i,k)$

Refine GE Algorithm (2)

- **Last version**
  - for $i = 1$ to $n-1$
    - for $j = i+1$ to $n$
      - $m = A(j,i)/A(i,i)$
      - for $k = i+1$ to $n$
        - $A(j,k) = A(j,k) - m \cdot A(i,k)$

Refine GE Algorithm (3)

- **Last version**
  - for $i = 1$ to $n-1$
    - for $j = i+1$ to $n$
      - $m = A(j,i)/A(i,i)$
      - for $k = i+1$ to $n$
        - $A(j,k) = A(j,k) - m \cdot A(i,k)$
  - for $i = 1$ to $n-1$
    - for $j = i$ to $n$
      - $A(j,i) = A(j,i)/A(i,i)$
      - for $k = i$ to $n$
        - $A(j,k) = A(j,k) - A(j,i) \cdot A(i,k)$

Refine GE Algorithm (4)

- **Last version**
  - Express using matrix operations (BLAS)
    - for $i = 1$ to $n-1$
      - for $j = i+1$ to $n$
        - $A(j,i+1:n) = A(j,i+1:n) - A(j,i) \cdot A(i,i+1:n)$

What GE really computes

- Call the strictly lower triangular matrix of multipliers $M$, and let $L = I+M$
- Call the upper triangle of the final matrix $U$
- **Lemma (LU Factorization):** If the above algorithm terminates (does not divide by zero) then $A = L\cdot U$
- Solving $A'x=b$ using GE
  - Factorize $A = L\cdot U$ using GE
  - Solve $L'y = b$ for $y$, using substitution
  - Solve $U'x = y$ for $x$, using substitution
  - Thus $A'x = (L'\cdot U')x = L'y = b$ as desired
Problems with basic GE algorithm

- What if some A(i,i) is zero? Or very small?
  - Result may not exist, or be “unstable”, so need to pivot

- BLAS 3 (matrix multiply) is fastest (Lecture 2)

```
for i = 1 to n-1
  A(i+1:n,i) = A(i+1:n,i) / A(i,i)         … BLAS 1 (scale a vector)
  A(i+1:n,i+1:n) = A(i+1:n , i+1:n )  … BLAS 2 (rank-1 update)
    - A(i+1:n , i) * A(i , i+1:n)
```

Peak
BLAS 3

BLAS 2

BLAS 1

Pivoting in Gaussian Elimination

- A = [0 1] fails completely, even though A is “easy”
- I illustrate problems in 3-decimal digit arithmetic:
  \[
  A = \begin{bmatrix}
  1e-4 & 1 \\
  1 & 1
  \end{bmatrix}
  \quad \text{and} \quad
  b = \begin{bmatrix}
  1 \\
  2
  \end{bmatrix}
  \]
  - correct answer to 3 places is x = [1]

- Result of LU decomposition is
  \[
  L = \begin{bmatrix}
  1 & 0 \\
  [R(1:4-1) & 1e4 & 1 \\
  0 & [R(1-1e4*1) & 0 & -1e4
  \]
  \]
  - No roundoff error yet
  - Error in 4th decimal place

- Check if A = L*U = [1e-4 1]
  - (2,2) entry entirely wrong

- Algorithm “forgets” (2,2) entry, gets same L and U for all |A(2,2)|<5

- Numerical instability
  - Computed solution x totally inaccurate
  - Cure: Pivot (swap rows of A) so entries of L and U bounded

Gaussian Elimination with Partial Pivoting (GEPP)

- Partial Pivoting: swap rows so that each multiplier
  \[|L(i,j)| = |A(j,i)/A(i,i)| \leq 1\]

```
for i = 1 to n-1
  find and record k where |A(k,i)| = max_{i \leq j \leq n} |A(j,i)|
  if |A(k,i)| = 0
    exit with a warning that A is singular, or nearly so
  else
    swap rows i and k of A
  end if
  A(i+1:n,i) = A(i+1:n,i) / A(i,i)        … each quotient lies in [-1,1]
  A(i+1:n,i+1:n) = A(i+1:n , i+1:n ) - A(i+1:n , i) * A(i , i+1:n)
```

° Lemma: This algorithm computes A = P’L’U’, where P is a
  permutation matrix

° Since each entry of |L(i,j)| <= 1, this algorithm is considered
  numerically stable

° For details see LAPACK code at www.netlib.org/lapack/single/sgetf2.f

Converting BLAS2 to BLAS3 in GEPP

- Blocking
  - Used to optimize matrix-multiplication
  - Harder here because of data dependencies in GEPP

- Delayed Updates
  - Save updates to “trailing matrix” from several consecutive BLAS2
    updates
  - Apply many saved updates simultaneously in one BLAS3
    operation

- Open questions remain
  - Need to choose a block size b
    - Algorithm will save and apply b updates
    - b must be small enough so that active submatrix consisting of b
      columns of A fits in cache
    - b must be large enough to make BLAS3 fast

Overview of LAPACK

° Standard library for dense/banded linear algebra
  - Linear systems: A\*x = b
  - Least squares problems: min_{x} || A\*x - b ||^2
  - Eigenvalue problems: Ax = \lambda x, Ax = \lambda Bx
  - Singular value decomposition (SVD): A = U \* V^T

° Algorithms reorganized to use BLAS3 as much as possible
  - Basis of math libraries on many computers
  - Many algorithmic innovations remain

° Projects available
Parallelizing Gaussian Elimination

- **Parallelization steps**
  - Decomposition: identify enough parallel work, but not too much
  - Assignment: load balance work among threads
  - Orchestrate: communication and synchronization
  - Mapping: which processors execute which threads

- **Decomposition**
  - In BLAS 2 algorithm nearly each flop in inner loop can be done in parallel, so with \( n^2 \) processors, need 3n parallel steps
  - This is too fine-grained, prefer calls to local matmul instead

Assignment of parallel work in GE

- Think of assigning submatrices to threads, where each thread responsible for updating submatrix it owns
- “owner computes” rule natural because of locality
- What should submatrices look like to achieve load balance?

Different Data Layouts for Parallel GE (on 4 procs)

- Best load balance:
- 0 1 2 3
- Load balanced, but can’t easily use BLAS2 or BLAS3

- Column Blocked Layout
- Column Cyclic Layout
- Can trade load balance for locality

The winner!

- Columns Block Cyclic Layout
- Row and Columns Block Cyclic Layout

- Block Skewed Layout

Blocked Partitioned Algorithms

- Orthogonal reduction to:
  - (upper) Hessenberg form
  - Symmetric tridiagonal form
  - Bidiagonal form
- Block QR iteration for nonsymmetric eigenvalue problems
- LR, QR, LQ factorizations
- Form Q or Qᵀ

Memory Hierarchy and LAPACK

- \( ijk \) implementations
- for \( _i = 1:n \)
  - Effects order in which data referenced; some better at allowing data to keep in higher levels of memory hierarchy
- for \( _j = 1:n \)
- for \( _k = 1:n \)
  - \( u_j \) \( a_{ik} \) \( w_{ij} \) \( v_{ji} \)
- end
- end

- Applies for matrix multiply, reductions to condensed form
  - May do slightly more
  - Up to 3 times faster

Derivation of Blocked Algorithms

**Cholesky Factorization** \( A = UᵀU \)

- Equating coefficient of the \( j^{th} \) column, we obtain
  - \( a_j \) \( U_jᵀu_j \)
  - \( a_j \) \( u_jᵀu_j \) \( u_j \)

- Hence, if \( U_j \) has already been computed, we can compute \( u_j \) and \( a_j \) from the equations:

- \( U_jᵀa_j \) \( u_j \)
- \( u_jᵀa_j \) \( a_j \)
Here is the body of the LINPACK routine SPOFA which implements the method:

```fortran
DO 30 J = 1, N
   INFO = J
   S = 0.0E0
   JM1 = J - 1
   IF( JM1.LT.1 ) GO TO 20
   DO 10 K = 1, JM1
      T = A( K, J ) - SDOT( K-1, A( 1, K ), 1,A( 1, J ), 1 )
      T = T / A( K, K )
      A( K, J ) = T
      S = S + T*T
   10 CONTINUE
20       CONTINUE
   S = A( J, J ) - S
   C ...EXIT
   IF( S.LE.0.0E0 ) GO TO 40
   A( J, J ) = SQRT( S )
30 CONTINUE
```

This change by itself is sufficient to significantly improve the performance on a number of machines.

From 72 to 251 Mflop/s for a matrix of order 500 on one processor of a CRAY Y-MP.

However on 378 Mflop/s on 8 Procs. Of a CRAY Y-MP.

Suggest further work needed.

Derivation of Blocked Algorithms

Equating coefficient of second block of columns, we obtain

\[ A_{22} = U_{21}^T U_{12} \]

Hence, if \( U_{11} \) has already been computed, we can compute \( U_{12} \) as the solution of the following equations

by a call to the Level 3 BLAS routine STRSM:

\[ U_{12}^T U_{12} = A_{22} \]

LAPACK Blocked Algorithms

```fortran
DO 10 J = 1, N, NB
   CALL STRSM( 'Left', 'Upper', 'Transpose','Non-Unit', J-1, JB, ONE, A, LDA, $               A( 1, J ), LDA )
   CALL SSYRK( 'Upper', 'Transpose', JB, J-1,-ONE, A( 1, J ), LDA, ONE, $               A( J, J ), LDA )
   CALL SPOTF2( 'Upper', JB, A( J, J ), LDA, INFO )
   IF( INFO.NE.0 ) GO TO 20
10 CONTINUE
```

On Y-MP, L3 BLAS squeezes a little more out of 1 proc, but makes a large improvement when using 8 procs.

LAPACK Ongoing Work

- Add functionality
  - updating/downdating, divide and conquer least squares, bidiagonal bisection, bidiagonal inverse iteration, band SYD, Jacobi methods, ...
- Move to new generation of high performance machines
  - IBM SPs, CRAY T3E, SGI Origin, clusters of workstations
- New challenges
  - New languages: FORTRAN 90, HP FORTRAN, ...
  - (CMMD, MPL, NX ...)
  - many flavors of message passing, need standard (PVM, MPI): BLACS
- Highly varying ratio
  - Computational speed
  - Communication speed
- Many ways to layout data,
- Fastest parallel algorithm sometimes less stable numerically.

LAPACK Contents

- Combines algorithms from LINPACK and EISPACK into a single package. User interface similar to LINPACK.
- Built on the L, 1, 2 and 3 BLAS, for high performance (manufacturers optimize BLAS)
- LAPACK does not provide routines for structured problems or general sparse matrices (i.e sparse storage formats such as compressed-row, -column, -diagonal, skyline ...).
History of Block Partitioned Algorithms
- Early algorithms involved use of small main memory using tapes as secondary storage.
- Recent work centers on use of vector registers, level 1 and 2 cache, main memory, and "out of core" memory.

Blocked Partitioned Algorithms
- Orthogonal reduction to:
  - (upper) Hessenberg form
  - symmetric tridiagonal form
  - bidiagonal form
- Block QR iteration for nonsymmetric eigenvalue problems
- LU Factorization
- Cholesky factorization
- Symmetric indefinite factorization
- Matrix inversion
- QR, QL, RQ, LQ factorizations
- Form Q or Q'T